A Higher-Order Extension for Imperative Synchronous Languages

Eric Vecchié, Jean-Pierre Talpin, Sébastien Boisgérault

A Higher-Order Extension for Imperative Synchronous Languages

Eric Vecchié
Mines ParisTech
60 Bd St Michel
FR-75272 Paris Cedex 06
Eric.Vecchie@ensmp.fr

Jean-Pierre Talpin
INRIA Rennes - Bretagne Atlantique
Campus de Beaulieu
FR-35042 Rennes Cedex
Jean-Pierre.Talpin@irisa.fr

Sébastien Boisgérault
Mines ParisTech
60 Bd St Michel
FR-75272 Paris Cedex 06
Sebastien.Boisgerault@ensmp.fr

ABSTRACT

This article presents the very first effective design of higher-order modules in the synchronous programming language Esterel. Higher-order modules, together with the robust separate compilation scheme that implements it, allow us to address a yet unexplored application spectrum ranging from rapid prototyping of embedded functionality to hot reconfiguration of embedded software within the formal modeling framework of the "*synchronous hypothesis*". While extensions of data-flow synchronous languages had already been proposed for Lustre [11] and Signal [25], the adaptation of similar programming concepts to imperative synchronous frameworks like Esterel has long posed major technical challenges, due to the specificity of its model of computation.
We present a framework including a formal semantics, a type system, and a modular code generator, that tackle this challenge. We consider a specific stack-based module call convention and a simple event pooling protocol ; in consequence signals can refer to modules and modules can be transmitted and instantiated by referencing a signal. We define a type system that computes the potential emissions of a module and prove it sound. Our type system seamlessly fits an extension of Esterel's constructive semantics with higher-order modules.

1. INTRODUCTION

Synchronous programming languages are domain-specific languages dedicated to the design of real-time embedded systems. Their semantics are based on the synchronous hypothesis stating that the execution of a program is discretely divided into atomic reactions. Enjoying efficient compilation methods, verification and automatic distribution tools, these languages are now employed in various industrial design processes (avionics, nuclear plants...). In this domain, the imperative style of Esterel [5] has been specifically designed to ease programming of control-dominated applications.

Basically, an Esterel module corresponds to a finite state machine reading input values and producing values on communication channels called *signals*. Therefore, as a *first-order* language, it is traditionally not possible to communicate modules on signals in Esterel. Modern real-time embedded systems however, often require to be dynamically reconfigurable, that is to be able to change parts of code to be executed at run-time. Concrete applications can be found in various domains like network switches, space missions, and more generally domains where quality of service must often be adapted with respect to the environment. A more concrete example concerns authentication and security systems in mobile telephony: authentication features in cellular phones are both heterogeneous and likely to evolve [23]. To ensure functioning of the phone, the system has then to reconfigure its identification protocol dynamically with respect to the nearest base station.

The increasing popularity of reconfigurable systems comes naturally with the increasing need of tools able to design them. In the context of synchronous languages, the possibility to write *higher-order* modules appears then as a natural answer to these requirements. Higher-order were successfully introduced into data-flow synchronous languages [11, 25, 13]. We propose such an extension for the imperative style of the synchronous language Esterel. More precisely, we address the problem of the constructiveness in the context of a modular execution scheme. Indeed, modularity in the sense of a separate compilation and execution of modules is a major issue in Esterel and also the key for the introduction of higher-order. For this purpose, we seamlessly extend the constructive semantics of Esterel with a single rule expressing the instantiation of a module from the value of a signal. To deal with constructiveness issues, we then define and show the correctness of a type system which allows us to know the potential emissions of a higher-order module.

1.1 Related Work

Higher-order features for concurrent systems have been extensively studied in the context of process calculi [19]. These calculi provide very powerful features like dynamic creation of processes and migration using a small collection of primitives. In some of these calculi, communication channels themselves can be sent through other channels, allowing the topology of process interconnections to change. Unlike Esterel, process calculi are asynchronous and nondeterministic and their model of computation are conflicting with the synchronous model we are considering here.

Some closely related approaches are based on the synchronous model. The reactive approach [9] provide a way to dynamically create new processes. The ULM language [7] combines the synchronous approach with the mobility feature inspired by the process calculi. In these approaches however the reaction to the absence of a signal are delayed to the next reaction. In Esterel the absence of a signal is likely to trigger a reaction in the same atomic instant. This immediate reaction to absence is at the heart of the constructive semantics of Esterel. It is also the main reason making its modular compilation difficult.

Today, solutions for programming reconfigurable systems are mainly based on middleware approaches [2, 10, 20]. The

SCOPES '10, June 28-29, 2010, St. Goar, Germany

main drawback of such approaches is the lack of high-level constructs and fine control on the behavior of the running program. As an example, let us consider a module M that has to be dynamically replaced by a module M'. Let us suppose that M' requires an initialization phase running over several reactions (or suppose that M requires a finalization phase to reach a stable state). If M/M' is a module responsible for some critical task and must not be interrupted, then M must keep on running until M' becomes fully operational. This simple example illustrates the fact that dynamic reconfiguration can rely on some strong interactions with the running program. Middleware solutions can then be arbitrarily complicated or even impossible to program whereas a higher-order based designs remain simple.

1.2 Modularity

The introduction of higher order in synchronous languages is made possible by the separate compilation and execution of synchronous modules. We start from the very same notion of modular compilation as in general-purpose languages like C: the translation of an Esterel module into binary code with a header file describing its interface. The instantiation of a module is simply realized by passing its parameters and executing its code through simple calling conventions. Our implementation is based on the key idea of defining concurrent coroutines and on a minimalist runtime system to execute separately compiled modules.

Early Work. Because it was thought to make causality analysis more difficult, modular compilation was not considered in early implementations of Esterel. Instead, early Esterel compilers generate code from a data-structure representing a flattened and inlined expansion of the source program. For instance, [6] translates a program into an extended finite state machine whose transitions are decorated with code fragments. The obvious disadvantage of this technique is the potential state-space explosion of the automaton. Its main advantage is the very short execution time of the generated code. Compilation techniques achieving a polynomial space complexity were first obtained by using systems of equations to symbolically represent the automaton of a program [4]. This approach was successfully used for hardware synthesis and it is still at the core of commercial tools, although the generated software is sometimes slower. Another approach is based on the translation of programs into concurrent control flow graphs [15, 21] whose sizes is in linear proportion of the given program. At each instant, the control flow graph is visited until active nodes are found to trigger the execution of the corresponding sub-tree. The approach followed by the Saxo-RT compiler [12] translates the program into an event graph.

Recent Work. Some works [24, 26] address different notions of "modular compilation". In [26], the generated code is able to partially deal with undefined inputs by using a three-valued logic. The generated program tries to compute as many outputs as it can while ignoring inputs that are still unknown. In [24], the Esterel dialect Quartz is translated into a "job language". This translation can be conceptually regarded as a graph of simultaneously active jobs where each job contains an atomic task. In these approaches, the increased effort involved for the separate compilation of modules separately does not spare "tuning" the generated code where it is used and depending on the calling context. Unlike our approach, it implies that the code of instantiated modules must be duplicated. Some recent works [18] use multi-function interfaces to generate modular code for synchronous block diagrams. Each function in the interface is responsible for the evaluation of some outputs through the evaluation of the relevant part of the module. The trade-off between modularity (decreasing with the number of functions in the interface) and reusability (increasing with the number of functions) is also addressed. Earlier works on Signal and Cronos [3, 16] actually use very similar techniques of clustering, interface synthesis and scheduling to perform modular compilation. In our approach, modularity is maximized since the compilation of each module generates a single function. Reusability is nevertheless ensured by a flexible model of execution which allows causality issues to be resolved dynamically at run-time.

1.3 Syntax of Esterel

An Esterel module consists of an interface where *input* and *output* signals are defined, followed by a body. Syntax of program statements is provided by the following simple grammar :

$$
\begin{array}{rcll}
P & ::= & \texttt{pause} & \mid \texttt{signal } S \texttt{ in } P \texttt{ end} \\
 & \mid & S := E & \mid \texttt{if } E \texttt{ then } P \texttt{ else } P \\
 & \mid & P \ ; \ P & \mid \texttt{abort } P \texttt{ when } E \\
 & \mid & P \parallel P & \mid \texttt{suspend } P \texttt{ when } E \\
 & \mid & \texttt{loop } P \texttt{ end} &
\end{array}
$$

with S ranging over signals and E ranging over expressions. Naive semantics of Esterel goes as follows : programs behaviors are discretely divided into instants or reactions. Control threads are executed until reaching a `pause` statement, which is the main statement which cuts behaviors into atomic instants. In a reaction *cycle*, input signals are read/sampled, and internal computation takes place until output signals are emitted in answer, and the program state is progressed. Instants are based on a *common* logical clock, which paces all parallel threads. Of course in a reaction various parallel threads do **not** run independently, as they may synchronize and affect one another causally (hardware people would say "combinationally"). When control reaches a `present(S)` boolean condition (testing signal presence at the current instant), it may have to postpone execution until a consistent definitive value and status are obtained for the signal inside the current reaction (either because it is emitted somewhere in parallel, or because other threads of execution progressed to a point where provably *all* potential emissions were discarded).

Example. Dynamically reconfigurable systems can be expressed in the syntax of Esterel as in the following example:

```
module RECONF(M : mod(...), params...)
  loop
    abort
      run (? M)(params...); halt
    when present(M)
  end
end module
```

where "`halt`" is syntactic sugar for "`loop pause end`". In this example, a module `M` is given as input of the module `RECONF`. The value of the module to be executed is carried by an input signal. This module is started/restarted each time the signal is emitted (each time its value changes). The module `RECONF` can thus be executed in parallel to another one responsible for choosing the module to be executed.

The paper is organized as follows: section 2 describes the considered constructive semantics of Esterel. The section 3 gives the semantic rules for the introduction of higher-order modules and section 4 describes the type system we use to deal with constructiveness issues in Esterel. The design of our modular compiler is then described in section 5. Section 6 provides some benchmarks and we finally conclude this article by some future works.

2. THE SEMANTICS OF ESTEREL

We consider the logical behavioral semantics of Esterel given in [8] (refer to these works for further details). This semantics is equivalent to the "official" semantics given in [5] but relies on a tiny but sufficient simple subset of the language where loops, preemption and suspension are discarded. In essence, loops are captured by the semantics of the sequence and preemption and suspensions are captured by the one of the test. Furthermore, constructiveness issues are more clearly addressed in this formalism. A process is interpreted by a series of transitions whose triggers are governed by rules of structured operational semantics. A run:

$$P \xrightarrow[I_1]{O_1} P_1 \dots P_{n-1} \xrightarrow[I_n]{O_n} P_n$$

depicts the successive reactions of a process P to the inputs $I_{1..n}$ producing the outputs $O_{1..n}$. The meaning of a transition is defined by an auxiliary chain of uncompleted reactions subject to SOS rules:

$$P \xrightarrow[I]{O} Q \Leftrightarrow P, I \xrightarrow{\perp} P_1, E_1 \xrightarrow{\perp} \dots P_m, E_m \xrightarrow{k} Q, I \cup O$$

where $k \in \{0, 1\}$. The status $\perp$ means that the execution of the current reaction is not completed. At the end of a reaction, the status k is either 0 to mean stopped or 1 for waiting. We call I, O or E an environment that maps names to values and P, Q a process. We have the following axioms and rules [5]:

- Nothing:

$$\textbf{\textit{nothing}}, E \xrightarrow{0} \textbf{\textit{nothing}}, E$$

- Pause:

$$\textbf{\textit{pause}}, E \xrightarrow{1} \textbf{\textit{nothing}}, E$$

- Assignment (emission of values). As we consider valued signals here and for simplification issue, we only consider programs where each signal is emitted once in each atomic reaction:

$$s\textbf{:=}v, E \cup \{s^{\perp}_{v'}\} \xrightarrow{0} \textbf{\textit{nothing}}, E \cup \{s^{+}_{v}\}$$

  where v is the value and b is the status ($+$ for *present*, $-$ for *absent* or $\perp$ for *undefined*) of a signal s^b_v.

- Sequence:

$$\frac{P, E \xrightarrow{k} P', E' \quad k \neq 0}{P\,\textbf{;}\,Q, E \xrightarrow{k} P'\,\textbf{;}\,Q, E'}$$

$$\frac{P, E \xrightarrow{0} P', E'}{P\,\textbf{;}\,Q, E \xrightarrow{\perp} Q, E'}$$

- Tests:

$$\frac{s^{+}_{v} \in E}{\textbf{\textit{if}}\ s\ \textbf{\textit{then}}\ P\ \textbf{\textit{else}}\ Q, E \xrightarrow{\perp} P, E}$$

$$\frac{s^{-}_{v} \in E}{\textbf{\textit{if}}\ s\ \textbf{\textit{then}}\ P\ \textbf{\textit{else}}\ Q, E \xrightarrow{\perp} Q, E}$$

- Parallelism:

$$\frac{P, E \xrightarrow{k} P', E'}{P \perp ||_l Q, E \xrightarrow{\perp} P' \,_k||_l Q, E'}$$

$$\frac{Q, E \xrightarrow{l} Q', E'}{P \,_k||_{\perp} Q, E \xrightarrow{\perp} P \,_k||_l Q', E'}$$

$$\frac{k \neq \perp \quad l \neq \perp}{P \,_k||_l Q, E \xrightarrow{\max(k,l)} P \perp ||_{\perp} Q, E}$$

- Local signals:

$$\frac{P, E \cup \{\Pi(s^b_v, P)\} \xrightarrow{\perp} P', E' \cup \{s^{b'}_{v'}\}}{\textbf{\textit{signal}}\ s^b_v\ \textbf{\textit{in}}\ P, E \xrightarrow{\perp} \textbf{\textit{signal}}\ s^{b'}_{v'}\ \textbf{\textit{in}}\ P', E'}$$

$$\frac{P, E \cup \{\Pi(s^b_v, P)\} \xrightarrow{k} P', E' \cup \{s^{b'}_{v'}\} \quad k \neq \perp}{\textbf{\textit{signal}}\ s^b_v\ \textbf{\textit{in}}\ P, E \xrightarrow{k} \textbf{\textit{signal}}\ s^{\perp}_{v'}\ \textbf{\textit{in}}\ P', E'}$$

  where $\Pi(s^b_v, P)$ depends on the status of the signal s or on $\pi(P)$, the set of potentially emitted signals in P:

$$\Pi(s^b_v, P) = \begin{cases} s^{-}_{\perp} \text{ if } b = \perp \text{ and } s \notin \pi(P) \\ s^b_v \text{ otherwise} \end{cases}$$

2.1 Constructiveness

Several variations of the constructive semantics of Esterel are presented in [8]. The semantics can thus be more or less restrictive depending on the way the so-called "*Potential Function*" $\pi(P)$ is defined. We consider the v3 version of the constructive semantics where $\pi(P)$ is defined by:

- Nothing, pause:

$$\pi(\textbf{\textit{nothing}}) = \pi(\textbf{\textit{pause}}) = \varnothing$$

- Assignment:

$$\pi(s\textbf{:=}v) = \{s\}$$

- Sequence:

$$\pi(P\,\textbf{;}\,Q) = \begin{cases} \pi(P) \cup \pi(Q) \text{ if } \gamma(P) \\ \pi(P) \text{ otherwise} \end{cases}$$

- Tests:

$$\pi(\textbf{\textit{if}}\ s\ \textbf{\textit{then}}\ P\ \textbf{\textit{else}}\ Q) = \pi(P) \cup \pi(Q)$$

- Parallelism:

$$\pi(P \,_k||_l Q) = \begin{cases} \pi(P) \cup \pi(Q) \text{ if } k = l = \perp \\ \pi(P) \text{ if } k = \perp \text{ and } l \neq \perp \\ \pi(Q) \text{ if } k \neq \perp \text{ and } l = \perp \\ \varnothing \text{ otherwise} \end{cases}$$

- Local signals:

$$\pi(\textbf{\textit{signal}}\ s^b_v\ \textbf{\textit{in}}\ P) = \pi(P) \setminus \{s\}$$

The predicate $\gamma(P)$ is *true* if the execution of P can be instantaneous. We have:

- Nothing, assignment:

$$\gamma(\textbf{\textit{nothing}}) \qquad \gamma(s\textbf{:=}v)$$

- Sequence:

$$\frac{\gamma(P) \quad \gamma(Q)}{\gamma(P\,\textbf{;}\,Q)}$$

- Tests:

$$\frac{\gamma(P)}{\gamma(\textbf{\textit{if}}\ s\ \textbf{\textit{then}}\ P\ \textbf{\textit{else}}\ Q)} \qquad \frac{\gamma(Q)}{\gamma(\textbf{\textit{if}}\ s\ \textbf{\textit{then}}\ P\ \textbf{\textit{else}}\ Q)}$$

- Parallelism:

$$\gamma(P\,{}_0||_0\,Q)$$

$$\frac{\gamma(P)}{\gamma(P\,{}_\perp||_0\,Q)} \qquad \frac{\gamma(Q)}{\gamma(P\,{}_0||_\perp\,Q)} \qquad \frac{\gamma(P)\quad\gamma(Q)}{\gamma(P\,{}_\perp||_\perp\,Q)}$$

- Local signals

$$\frac{\gamma(P)}{\gamma(\textbf{\textit{signal}}\ s_v^b\ \textbf{\textit{in}}\ P)}$$

The correctness of this constructive semantics, i.e the property intuitively stating that "*all signals that are emitted are indeed detected*", were also proven in [8].

3. HIGHER-ORDER EXTENSION

A higher-order extension of Esterel requires the addition of the following simple rule stating that a module whose value is carried by a signal can be instantiated through it:

$$\frac{s_v^b \in E \quad b \neq \perp \quad v = \textbf{\textit{module}}(s_1, \ldots s_n)\,.\,P}{\textbf{\textit{run}}\ (?s)(v_1, \ldots v_n), E \xrightarrow{\perp} P\,[v_1, \ldots v_n/s_1, \ldots s_n], E}$$

Without further information, the computation of possibly emitted signal in such statements has to consider the "worst case" where any parameter of the module can be emitted and where the execution of the module can be instantaneous. The value of the potential function for higher-order instantiation of modules is then:

$$\pi(\textbf{\textit{run}}\ (?s)(v_1, \ldots v_n)) = \{v_1, \ldots v_n\}$$

and we have:

$$\gamma(\textbf{\textit{run}}\ (?s)(v_1, \ldots v_n))$$

A better solution is to capture this information in a type system which flags for each module:

1. the output signals of a module that can possibly be emitted when the module is started ($\pi(P)$),
2. if the execution of the module can be instantaneous ($\gamma(P)$).

Closures. Esterel modules are traditionally declared at the top-level. It is nevertheless possible to introduce closures and to allow nested declarations of modules as it is the case in "traditional" functional languages:

```
module EXT(I: sig, M: mod(sig))

  module INT(O: sig)
    await I;
    emit O
  end module

  M := INT
end module
```

The "`emit`" operator is syntactic sugar for the emission of pure signals. In this example, the signal `I` is captured by the nested module `INT`. This module is then emitted on `M` so that `I` escapes and remains accessible in `M`. As the correct execution of programs requires the knowledge of potentially emitted signals $\pi(P)$, the emission of any "external" signal should be prohibited at the first reaction of any module. This is not the case for the next reactions since the body of the module is then "expanded".

Causality Issues. Higher-order offers a way to transmit modules on signal, including modules that are written after the main program has been started. Causality in the context of higher-order is not a static issue. In our approach, causality issues are therefore resolved at run-time. Our major concern is the correct execution of the programs, that is the correct ordering of instructions. Let us consider the following example:

```
    1 if present(I)  then emit S1;
    2 if present(S2) then emit O
||
    3 if present(S1) then emit S2
```

The correct execution of this program starts with the beginning of the first thread (at line 1), then carry on to the second one (at line 3) and finally resume the first one (at line 2). This program is a friendly illustration of the problem. In real life we might face some nastier configurations where such three conditional statements are nested in different higher-order modules. Section 5 provides practical solutions to this problem.

In Esterel, the composition of two correct modules may generate a causally incorrect program. This is typically the case in the following program:

```
present(S) then nothing else emit S
```

where `S` is absent if it emitted and present if it is not: this is incorrect in Esterel. In our approach, this problem is related to the presence of deadlocks in the program. Code generation and rejection of causally incorrect programs should be independent tasks. The detection of causally incorrect programs is a major concern that deserves a dedicated article but it is not in the scope of this one: in our approach, we guarantee the correct execution of higher-order programs under the hypothesis that the programs are causally correct. In our compiler however, deadlocks are easily detected at run-time.

4. TYPE SYSTEM

As a motivating example, let us consider the following simple Esterel program:

```
module MOD(M, S1, S2)
  if present(M) then
    if present(S1) else
      run (? M)(S1, S2)
    end if
  end if
end module
```

So as to remain constructively correct, the module passed on `M` is not allowed to emit `S1` in the first instant. We write:

```
mod(sig, Π sig)
```

the type expected for the signal `M` where the flag "Π" indicates that the signal can be emitted immediately. We also have subtyping since a module of type:

```
mod(sig, sig)
```

could also be received on `M` without problem. Let us now consider the following examples:

```
module A0(s: sig)
  pause; emit s
end module

module A1(s: Π sig)
  emit s; pause
end module
```

The type of `A0` is "smaller" than the type of `A1` in the sense that you can use `A0` each time you expect `A1`.

```
module B0(a: mod(sig), s: sig)
  run (? a)(s)
end module

module B1(a: mod(Π sig), s: sig)
  pause; run (? a)(s)
end module
```

We have `B1` $\sqsubseteq$ `B0` (Contravariance).

```
module C0(a: mod(sig))
  a := A0
end module

module C1(a: mod(Π sig))
  a := A1
end module
```

We now have `C0` $\sqsubseteq$ `C1` (Covariance). In the case of `B0` and `B1`, signal `a` is used as an input: the subtyping relation is contravariant. In the case of `C0` and `C1`, signal `a` is used as an output: the subtyping relation is covariant. In our type system, we then choose to make the distinction between the input type and the output type of signals.

4.1 Definition

As we want to allow subtyping, our type system has the structure of a lattice and the type t of a module is defined by:

$$\begin{array}{rcl} t & ::= & m \mid \top \mid \bot \\ m & ::= & [\Gamma]\ \mathsf{mod}([\Pi]\ s, \ldots [\Pi]\ s) \\ s & ::= & \mathsf{sig\ of}(\mathsf{in} : t, \mathsf{out} : t) \end{array}$$

where Γ and Π are the markers indicating respectively that a module can be instantaneous (Γ) and that a signal is possibly immediately emitted (Π). $\top$ and $\bot$ are respectively the greatest and the least module type such that $\forall t, t \sqsubseteq \top$ and $\bot \sqsubseteq t$. The parameters of a module are signal where value can flow from caller to callee and vice versa. For a matter of formalization, the type of a signal is then divided into its input type and its output type. This can also be seen as an assume/guarantee pair where:

1. the assumption is made that any value read on the signal is "smaller" than its input type,
2. the guarantee is given that any value emitted on the signal is "smaller" than its output type.

The subtyping relation between signals $S \sqsubseteq S'$ is covariant in the input and contravariant in the output. If we suppose the types S and S' to be:

$$\begin{array}{rcl} S & = & \mathsf{sig\ of}(\mathsf{in} : I, \mathsf{out} : O) \\ S' & = & \mathsf{sig\ of}(\mathsf{in} : I', \mathsf{out} : O') \end{array}$$

Then we have:

$$S \sqsubseteq S' \qquad \Leftrightarrow \qquad I \sqsubseteq I' \text{ and } O' \sqsubseteq O$$

This distinction between the input type and the output type of a signal does not mean that these types can be completely different. Indeed, for any signal of type:

$$\mathsf{sig\ of}(\mathsf{in} : I, \mathsf{out} : O)$$

there exists the implicit subtyping relation $O \sqsubseteq I$ as we show in section 4.3. If the type of a signal does not respect this constraint then this signal will be impossible to use concretely without breaking the type correctness of the program defined in section 4.2. In this system, an input signal written "`input s:`I" in the concrete Esterel syntax has then a type of the form:

$$\mathsf{sig\ of}(\mathsf{in} : I, \mathsf{out} : \bot)$$

An output signal written:

`output s:` O

has a type of the form:

$$\mathsf{sig\ of}(\mathsf{in} : \top, \mathsf{out} : O)$$

The subtyping relation for module types is contravariant. If we suppose the module types M and M' to be:

$$\begin{array}{rcl} M & = & i\ \mathsf{mod}(p_1\ T_1, \ldots p_n\ T_n) \\ M' & = & i'\ \mathsf{mod}(p'_1\ T'_1, \ldots p'_n\ T'_n) \end{array}$$

We then have $M \sqsubseteq M'$ if and only if:

$$\begin{array}{c} (i = \Gamma) \Rightarrow (i' = \Gamma) \quad \text{and} \\ \forall k \in [1..n] \quad (p_k = \Pi) \Rightarrow (p'_k = \Pi) \quad \text{and} \quad T'_k \sqsubseteq T_k \end{array}$$

The main advantage of using two different types for signals is illustrated by the following example where a module `M` using one formal parameter is declared:

```
module M(s: sig of(in : I, out : O))
  ...
end module
```

The couple of types I and O defines a range for possible use cases for `M`: with our type system, it is then possible to instantiate the module `M` with any signal of type T (i.e. $\mathsf{sig\ of}(\mathsf{in} : T, \mathsf{out} : T)$) as soon as T verifies $O \sqsubseteq T \sqsubseteq I$. This precision would be impossible to obtain with a unique type.

With such type system, the definition of the potential function for higher-order module instantiation can be refined. Thus, if the type of a signal s is:

$$s : \mathsf{sig\ of}(\mathsf{in} : I, \mathsf{out} : O)$$

where:

$$I = i\ \mathsf{mod}(p_1\ T_1, \ldots p_n\ T_n)$$

then we have:

$$\pi(\textbf{\textit{run (?s)}}(v_1, \ldots v_n)) = \{v_i \mid (p_i = \Pi)\}$$

and:

$$\frac{i = \Gamma}{\gamma(\textbf{\textit{run (?s)}}(v_1, \ldots v_n))}$$

The types $T_1, \ldots T_n$ of the parameters are supposed to be given by the programmer but the flags i and $p_1, \ldots p_n$ could be inferred easily. In future works we intend to infer more type information from the structure of the program and from a static analysis of its dependencies.

4.2 Type Correctness

Let $\mathcal{C}(D, P)$ be the predicate indicating that a program P is correctly typed given the environment D. An environment D is a list of declarations relating names x to types T. We write $D(x)$ for the type t associated with the name x in the environment D containing the declaration $[x : T]$:

$$D ::= \epsilon \mid D, [x : T]$$

By definition of $\mathcal{C}(D, P)$, the following properties hold:

- Nothing, pause:

$$\mathcal{C}(D, \textbf{\textit{nothing}}) \qquad \mathcal{C}(D, \textbf{\textit{pause}})$$

- Sequence:

$$\frac{\mathcal{C}(D,P) \quad \mathcal{C}(D,Q)}{\mathcal{C}(D, P\,\mathbf{;}\,Q)}$$

- Tests:

$$\frac{\mathcal{C}(D,P) \quad \mathcal{C}(D,Q)}{\mathcal{C}(D, \mathbf{if}\ s\ \mathbf{then}\ P\ \mathbf{else}\ Q)}$$

- Parallelism:

$$\frac{\mathcal{C}(D,P) \quad \mathcal{C}(D,Q)}{\mathcal{C}(D, P\ {}_k||_l\ Q)}$$

- Assignment. If we have:

$$\begin{array}{rcl} D(s) & = & \mathsf{sig\ of}(\mathsf{in} : I, \mathsf{out} : O) \\ D(v) & = & I' \quad \text{or} \quad D(v) = \mathsf{sig\ of}(\mathsf{in} : I', \mathsf{out} : O') \end{array}$$

then:

$$\frac{I' \sqsubseteq O}{\mathcal{C}(D, s\mathbf{:=}v)}$$

- Local signals. If we have:

$$D' \quad = \quad D, [s : \mathsf{sig\ of}(\mathsf{in} : T, \mathsf{out} : T)]$$

then:

$$\frac{\mathcal{C}(D', P)}{\mathcal{C}(D, \mathbf{signal}\ s : T\ \mathbf{in}\ P)}$$

- Module instantiation. If we have:

$$\begin{array}{rcl} D(s) & = & \mathsf{sig\ of}(\mathsf{in} : I, \mathsf{out} : O) \\ I & = & i\ \mathsf{mod}(p_1\ T_1, \ldots p_n\ T_n) \\ D(v_k) & = & T'_k \end{array}$$

then:

$$\frac{\forall k \in [1..n] \quad T'_k \sqsubseteq T_k}{\mathcal{C}(D, \mathbf{run}\ \mathbf{(?}s\mathbf{)(}v_1, \ldots v_n\mathbf{)})}$$

- Module declaration. If we have:

$$\begin{array}{rcl} D(m) & = & i\ \mathsf{mod}(p_1\ T_1, \ldots p_n\ T_n) \\ D' & = & D, [s_1 : T_1], \ldots [s_n : T_n] \end{array}$$

then:

$$\frac{\begin{array}{c}\gamma(P) \Rightarrow (i = \Gamma) \\ \forall k \in [1..n]\ s_k \in \pi(P) \Rightarrow (p_k = \Pi) \\ \mathcal{C}(D', P)\end{array}}{\mathcal{C}(D, \mathbf{module}\ m\ \mathbf{(}s_1 : T_1, \ldots s_n : T_n\mathbf{)}\ \mathbf{.}\ P)}$$

4.3 Soundness

As for the semantics, the type system is sound if all signals that are emitted are indeed detected:

$$\begin{array}{rl} \forall D : & \mathcal{C}(D,P) \quad \text{and} \quad P, E \xrightarrow{k} P', E' \\ \Rightarrow & \mathcal{C}\left(D, P'\right) \quad \text{and} \quad \forall s_v^+ \in E' - E \quad s \in \pi(P) \end{array}$$

This property were proven for the standard Esterel semantics in [8]. The introduction of higher-order does not change the structure of this proof but we have to prove additionally that a signal used for the instantiation of a module contains no "spurious" value. Concerning the type system, this comes down to say that all the potential emissions of a higher-order module being instantiated are indeed captured by the type system. In other words, we have to show that, given an environment for signal values E and an environment of declarations D, then the instantiation of a higher-order module $\mathbf{run\ (?}s\mathbf{)(}v_1, \ldots v_n\mathbf{)}$ always verifies:

$$\begin{array}{rcl} D(s) & = & \mathsf{sig\ of}(\mathsf{in} : I, \mathsf{out} : O) \\ I & = & i\ \mathsf{mod}(p_1\ T_1, \ldots p_n\ T_n) \\ s_v^b \in E \quad \text{and} \quad v & = & \mathbf{module(}s_1, \ldots s_n\mathbf{)}\ \mathbf{.}\ P \end{array}$$

$$\begin{array}{rl} & \gamma(P) \Rightarrow (i = \Gamma) \\ \forall k \in [1..n] & s_k \in \pi(P) \Rightarrow (p_k = \Pi) \end{array}$$

It is then sufficient to prove that, if a program P is correctly typed according to $\mathcal{C}(D,P)$, then the type of v is always "smaller" than the type I. We then have to prove by structural induction that for any signal s, any value emitted on s is "smaller" than any type expected on s. As a remark, we only consider "correct" programs where signals are always assigned before being read.

Sketch of Proof. We prove by induction that for any signal s, there exists a type T_0 such that T_0 is "smaller" than any expected type and "bigger" than any value emitted on s:

- The trivial case arise in the context of a local declaration:

$$\mathbf{signal}\ s : T\ \mathbf{in}\ P$$

In this case, the type system says:

$$D(s) = \mathsf{sig\ of}(\mathsf{in} : T, \mathsf{out} : T)$$

In this context we have $T_0 = T$ verifying the property for any correctly typed emission $s\mathbf{:=}v$ and any correctly typed reading $\mathbf{(?}s\mathbf{)}$ of s in P.

- By structural induction, a signal s can be passed as parameters to another module $\mathbf{run\ (?}m\mathbf{)(}s\mathbf{)}$. The type of s and m are supposed to be:

$$\begin{array}{rcl} D(s) & = & \mathsf{sig\ of}(\mathsf{in} : I, \mathsf{out} : O) \\ D(\mathbf{(?}m\mathbf{)}) & = & i_m\ \mathsf{mod}(p_m\ T_m) \end{array}$$

First, the property that the value of m is correct with respect to its type has to and can itself be proven by structural induction: more generally, the proof of the property must always consider signals of type $\mathsf{mod}(T)$ before signals of type T. As recurrence hypothesis, we then suppose we have T_0 such that $T_0 \sqsubseteq I$ and $O \sqsubseteq T_0$. If the expected type T_m for the parameter of $\mathbf{(?}m\mathbf{)}$ has the form:

$$T_m \quad = \quad \mathsf{sig\ of}(\mathsf{in} : I_m, \mathsf{out} : O_m)$$

then the type correctness rules imply that:

$$T \sqsubseteq T_m \quad \text{and then} \quad I \sqsubseteq I_m \ \text{ and } \ O_m \sqsubseteq O$$

We finally can prove that the property is also verified by structural induction:

$$T_0 \sqsubseteq I_m \quad \text{and} \quad O_m \sqsubseteq T_0$$

5. OUR COMPILER

We implemented a lightweight compiler able to deal with higher-order Esterel modules and complying with the constructive semantics given in section 2. The compilation method tries to mimic the well-known code generation techniques described in classical compilation books, like the well-known Dragon Book [1]. Expressions are flattened and translated into assembly code sequences. Conditional statements are replaced by conditional gotos and function calls use the

stack for passing the parameters and saving the register context of the caller.

In our approach, the execution scheme is inspired by the reactive kernel of Junior [17]. This tool is a relative of the Esterel language used to write reactive applications through a Java API and featuring a delayed reaction to absence. In Junior, each reaction step executes and reduces a tree representing the instantaneous state of the reactive program and reflecting its original structure. We use a mechanism similar to coroutines [14] so that the control yields at some points of the generated program. These coroutines are then hierarchically nested to reflect the control structure of the source program.

Runtime System. A Runtime System maintains a collection of *cooperative* threads to be executed during the current logical instant (reaction) and a second one to be executed at the next logical instant. The "*current*" list of threads can be dynamically enlarged. This is typically the case when a thread reaches a parallel statement. Thus, the execution of a reaction step means the execution of all the threads of the *current* list until it is made completely empty. The same way, the execution of a synchronous program means the execution of reaction steps until the "*next*" list of threads is empty. The second task of the Runtime System is also to deal with the signal emissions and resets but we shall come back to this later.

Our compiler generates sequential C code in assembly style (computed gotos, explicit stack manipulation,...). Each module in the input language is translated into a C function that can be executed through the Runtime System. Code generation rules are given in the following.

5.1 Sequential Statements

The compilation of classical sequential statements follows a classical compilation scheme [1] using conditional gotos for *if-then-else* and *do-while* statements. We shall simply consider these statements as part of our "high-level assembler". In the same way, local variables are referred through their names rather than FP[i] where FP is the Frame Pointer register and i would be the relative address of the concerned variable in the current frame. New stack frames are allocated when scopes are entered. In our implementation, a particular care has been taken to reallocate unused frames through a simple garbage collection mechanism.

Schizophrenia is a usual issue in the compilation of imperative synchronous programs [5]. It comes from the fact that, because of loops, several instances of a same variable can coexist simultaneously (in a same reaction step). The problem does not arise here thanks to the use of stack frames: each time a scope is re-entered a new data frame is allocated. The separation between control and data is at the heart of our modular compilation technique since it allows to simultaneously run several threads sharing the same code but working on distinct data.

5.2 Parallel Statements

In cooperative multithreading, each thread is responsible for relinquishing control. This is ensured by the assembly instruction "`stop`". Starting a new thread is done with the instruction "`start` *pc*, *fp*" where *pc* is the address of the first instruction of the started thread and *fp* is its frame pointer. The translation of a parallel statements P_1 `||` P_2 is then the following:

```
sync = 2
start FORK_lbl, FP

P1
goto SYNC_lbl

FORK_lbl:
  P2

SYNC_lbl:
  sync = sync-1
  if sync > 0 then
    stop
  endif
```

Here "`sync`" is a local variable used to synchronize P_1 and P_2 so that the first thread reaching `SYNC_lbl` is stopped and the second one goes on. This translation can be easily adapted for n-ary parallel statements.

5.3 Hierarchical Execution Scheme

The execution of a synchronous program is a succession of atomic reactions during which threads run in parallel until reaching a `pause` statement. In the beginning of each reaction step, the threads are resumed at the very locations where the program eventually paused at the end of the previous reaction. However, **not all** threads are resumed: because of `abort` and `suspend` statements, the resumption of threads should actually be performed hierarchically, according to the structure of the source program and the dynamically fulfilled conditions. In the following Esterel example:

```
abort
  suspend
    P
  when C2
when C1
```

the compilation into assembly code should provide a solution for hierarchically testing C_1 then C_2 before executing P, whatever it comprises of (involving complex parallel compositions and/or module instantiations).

5.3.1 Guarded "`pause`"

Each `abort` and `suspend` statement is then responsible for starting the threads running inside its body. In other words, each `pause` statement is guarded by the closest `abort` or `suspend` parent statement or, at the top level, by the Runtime System. The assembly code for `pause` statements is then the following:

```
  RPC = PAUSE_lbl
  RFP = FP
  FP  = FP[0]^n
  goto parent_guard_lbl
PAUSE_lbl:
```

where $\mathsf{FP[0]}^n = (\mathsf{FP}\underbrace{\mathsf{[0]})\ldots\mathsf{[0]}}_{n \text{ times}}$

The context of the thread is saved in two data registers: the resume point is saved in RPC (Resume Program Counter) and the current frame pointer in RFP (Resume Frame Pointer). The frame pointer is then popped of as many levels (given in n) as to retrieve the frame pointer of the guarding `abort`/`suspend` statement. At the address *parent_guard_lbl* is the code responsible for managing the paused thread. The values *parent_guard_lbl* and n are defined at compile time.

In the following, we shall use the simpler assembly instruction "`pause` *res*" where *res* is the address where the `pause` has to resume at the next reaction step. The "`pause` *res*" instruction is comparable to that of the `yield()` instruction of languages implementing coroutines [14] (Java, Python, Lua...). Thus, the previous code becomes simply:

```
  pause PAUSE_lbl
PAUSE_lbl:
```

5.3.2 Guard Statements

The task of `abort` and `suspend` statements comprises of guarding the execution of their bodies. Their compilation has to provide the assembly code for managing the nested `pause` statements through the callback mechanism described before and the data registers RPC and RFP. The compilation of `abort` statements produces the following code:

```
    P
    goto END_lbl

  GUARD_lbl:
    if child == [] then
      child = (RPC, RFP)::child
      pause RESUME_lbl
    else
      child = (RPC, RFP)::child
      stop
    endif

  RESUME_lbl:
    if not(C) then
      start_all(child)
      child = []
      stop
    endif

  END_lbl:
```

The assembler block starting at the addresses `RESUME_lbl` is responsible for resuming the execution of the paused threads inside P, under the condition C. Each time a thread of P is paused, it is added to a list "`child`" (a local variable stored in the current frame). These threads shall be passed through the registers RPC and RFP and managed from the address `GUARD_lbl`. This label has to be provided when compiling `pause` statements inside P, so that the control jumps to this very address each time a thread is paused. When the first thread is registered (`child == []`), the `abort` statement has also to register itself to the parent guard (`pause RESUME_lbl`). The translation of `suspend` statements is hardly different so that when the condition C holds, the paused threads are not started. Instead, the list is kept and the `suspend` statement registers itself again to the parent guard:

```
  RESUME_lbl:
    if C then
      pause RESUME_lbl
    else
      start_all(child)
      child = []
      stop
    endif
```

In the case of the `abort` statement, P is simply not restarted and the control flows to the rest of the program.

The translation of `weak abort` statements is slightly more complex since the abortion of P has to take place one logical instant later than it would be in the case of an `abort` statement. Nevertheless, this translation is similar enough to that of the `abort` statement for doing without such extra details here.

5.4 Modules

The instantiation of synchronous modules can be compiled as an *almost* classical function call where caller context and parameters are passed through a stack frame. The only difference with respect to the classical function call of sequential programs is that the guard context has to be stacked (since the module could be reentered several times along the successive reactions). The compilation of a module instantiation:

mod_name(p_1, ... p_m)

is then the following:

```
    SP = new_frame(m + 4)
    SP[0] = RETURN_lbl
    SP[1] = FP
    SP[2] = parent_guard_lbl
    SP[3] = FP[0]^n
    SP[1 + 3] = p_1
    ...
    SP[m + 3] = p_m
    FP = SP
    goto mod_name
  RETURN_lbl:
```

As for `pause` statements, the value of *parent_guard_lbl* and n are defined at compile time. As to cope with our calling conventions, the declaration of a module:

```
  module mod_name(S_1, ... S_n)
    P
  end module
```

generates the following code:

```
  mod_name:
    P
    ret = FP[0]
    FP = FP[1]
    goto ret

  GUARD_lbl:
    ret = FP[2]
    FP = FP[3]
    goto ret
```

Where `GUARD_lbl` is the parent guard label of the module body P. It is necessarily provided for the compilation of any `pause`, `abort` or `suspend` statement of P.

5.5 Dealing with Signals

Over the rigid framework provided by the structural translation of synchronous modules into cooperative sequential threads, we shall now provide a solution to deal with runtime causality induced by signals.

5.5.1 Implementation of Signals

At the end of a reaction step, the status of any signal has to be either *present* or *absent*. Inside a reaction, every signals but inputs remain *undefined* until reaching an `emit` statement or reaching a state of the program where all potential emissions are discarded.

We propose the following approach: before a signal is read or tested, we check its status. If it is *undefined*, then the execution of the thread is suspended. Each signal maintains a list of pending threads, so that they are immediately restarted as soon as a definitive value or status of the signal is determined (observer pattern). For this purpose, we use an assembly instruction "`wait` S", defined as a macro for the following code:

```
    if S.status == UNDEFINED then
      S.pending = (RESUME_lbl, FP)::S.pending
      stop
    endif
  RESUME_lbl:
```

The emission or absence of a signal is realized through the assembly instruction "`emit` S, *status*, *value*", defined as a macro for the following code:

```
S.status = status
if status == PRESENT then
  S.value = value
endif
start_all(S.pending)
```

5.5.2 Immediate Reaction to Absence

Reacting to the absence of a signal depends on the global behavior of the program. Our approach involves the Runtime System at the top level which maintains a list "*pending*" of undefined signals. At some point some signals have to be declared absent so as to achieve the current reaction step. However, **not all** the pending signals can be declared as absent. Let us consider the following example:

```
   if present(S1) then emit S3 else emit S2
||
   if present(S2) then emit S3
||
   if present(S3) then emit O
```

where `S1`, `S2` and `S3` are local signals. The execution of this program leads to a point where the threads are blocked on these signals. At this very point, only `S1` is safe to be set absent since `S2` and `S3` still have potential emitters.

5.5.3 Potential Emissions

Our solution relies on a local knowledge of the potentially emitted signals at each point of the program [22]. We use reference counters on signal to indicate which signal can be safely set absent. We introduce the instructions "`can` S" to mark a signal as possibly emitted in the current thread (S`.refcount = `S`.refcount+1`). We introduce the instruction "`cannot` S" to mark that the control just reached a point where a signal cannot be emitted any more in the current instant and by the current thread (S`.refcount = `S`.refcount-1`). This strategy complies with the definition of our *Potential Function* given in section 2.

5.5.4 Reference Counter Policy

At compile time, we **locally** compute the set of potentially emitted signals for each statement of the program. We then generate the `can` statements at each resumption point of the program (i.e for each `pause` statement), so that a thread immediately declares its potentially emitted signals as soon as it is started or resumed. Finally, we generate the `cannot` statements at each point of the target code where the set of potential emissions decreases. For example, the insertion of `can` and `cannot` statements in the code of the figure 1.a will produce the code of the figure 1.b.

```
pause

emit A

if present(S) then

  emit B

else

  emit C

pause
```

(a)

```
pause;
can A; can B; can C
emit A
cannot A
if present(S) then
  cannot C
  emit B
  cannot B
else
  cannot B
  emit C
  cannot C
pause
```

(b)

Figure 1: Insertion of `can` and `cannot` statements.

A signal can thus be safely declared absent when its reference counter is equal to zero and when the list of active threads is empty (which means that any active threads had the opportunity to increase the reference counter of the concerned signals before suspending their execution). The global information about the potentially emitted signals is thus obtained by the execution of all the active threads providing local partial information. This strategy for the generation of `can` and `cannot` instructions is not unique and leaves room for optimizations.

In the context of a modular execution, the correct compilation of programs requires some minimal knowledge about the modules. So as to identify the potentially emitted signals, we then need to know:

1. which parameters can be emitted at the first reaction of the module and
2. if the module can be instantaneous.

This information can be carried by a type system and stored in a header file. When a module is about to return, it also requires the list of signals that are potentially emitted by the caller after the termination of the callee. This information is actually passed on stack frames.

5.6 Higher-Order Modules

The main originality of this approach with respect to the related others lies in that the scheduling of the program blocks is not settled statically at compile time. Furthermore, the possibility of compiling synchronous modules as "black boxes" offers us the key for seamlessly introducing higher-order modules in imperative synchronous languages. For our compiler indeed, a module is a simple pointer Modules can thus be transmitted through signals and be instantiated from a signal value easily.

6. EXPERIMENTAL RESULTS

We developed a lightweight compiler based on these works called fnec that generates C-code. We use a standard extension to the C language known as "computed goto", where program labels can be handled as any `(void*)` pointers. We tested our compiler on many little examples and on the standard `wristwatch` example (see table 2). The generated code is quite big and slow, which is not a surprise considering that no effort was made for performance. The C code generation however is itself very fast.

These experimental results are more illustrative than relevant since our unique motivation was the introduction of higher-order modules in Esterel. In future works, efficient techniques [21, 15] shall be used to improve the execution speed of our modular code.

7. CONCLUSION

We presented an extension of the Esterel language for the introduction of higher-order modules. The benefit of it lies in its increased flexibility and its ability to better design synchronous systems. Constructiveness issues are managed by the mean of a type system indicating the potentially emitted signals of the modules. This extension has been implemented as a lightweight compiler generating sequential software (C) code.

As future works, we plan to use static analysis techniques to infer as much type information as possible. The idea would be to exploit a graph describing possible instantaneous dependencies between signals. Then, depending on the program locations where higher-order modules are instantiated, infer as much as possible which parameters are safe to be emitted in the first reaction and if the module can safely terminate instantaneously. We shall also investigate code migration as the next-step extension of Esterel. The principle would be to start a module somewhere in the program, then suspend its execution and resume it at another

example	size of files			compilation time		execution time
	source (.strl)	target (.c)	binary (.o)	.strl → .c (fnec)	.c → .o (gcc)	(10^6 reactions)
ABRO	0.20 KB	5.45 KB	8.73 KB	0 s	0.22 s	2.10 s
$A^{10}RO$	0.45 KB	8.51 KB	12.46 KB	0 s	0.30 s	1.98 s
arbiter4	1.04 KB	13.86 KB	23.09 KB	0 s	0.63 s	19.11 s
arbiter8	1.48 KB	19.91 KB	34.94 KB	0.01 s	1.03 s	36.81 s
wristwatch	25.1 KB	268.7 KB	276.7 KB	0.20 s	7.31 s	37.21 s

Figure 2: Experimental results on some Esterel programs

location in the code. Actually, such an extension would be quite easy to implement in our higher-order-tolerant compiler. The challenge lies mainly in finding the best (simplest) way to express migration in Esterel without conflicting with the existing constructs of the language. On a technical level, we also plan to consider dynamically reconfigurable components like FPGAs as a target for our higher-order Esterel compiler. The instantiation of a higher-order module would then be the result of an on-the-fly hardware copy of the instantiated module together with a connection of the parameters signals to the sockets of the module.

8. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1996.

[3] A. Benveniste, P. Le Guernic, and P. Aubry. Compositionality in dataflow synchronous languages: Specification & code generation. Research Report 3310, INRIA, November 1997.

[4] G. Berry. A hardware implementation of pure Esterel. In *Workshop on Formal Methods in VLSI Design*, Miami, Florida, 1991.

[5] G. Berry. The constructive semantics of pure Esterel. http://www-sop.inria.fr/esterel.org, July 1999.

[6] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[7] G. Boudol. ULM: a core programming model for global computing. In *European Symposium on Programming, ESOP'04*, Barcelona, Spain, April 2004.

[8] F. Boussinot. SugarCubes implementation of causality. Research Report 3487, INRIA, September 1998.

[9] F. Boussinot and J.-F. Susini. The SugarCubes tool box: a reactive Java framework. *Software-Practice and Experience*, 28(14):1531–1550, December 1998.

[10] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in Java. In *Int. Symposium on Component Based Software Engineering, CBSE'04*, Edinburgh, Scotland, May 2004.

[11] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Conference on Functional Programming (ICFP)*, pages 226–238, Philadelphia, USA, 1996. ACM.

[12] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil. SAXO-RT: Interpreting Esterel semantics on a sequential execution structure. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 65(5), 2002.

[13] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a higher-order synchronous data-flow language. In G. Buttazzo, editor, *International Conference on Embedded Software, EMSOFT'04*, pages 230–239, Pisa, Italy, September 2004. ACM, New-York.

[14] M. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7), July 1963.

[15] S. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 21(2):169–183, February 2002.

[16] O. Hainque, L. Pautet, Y. Le Biannic, and E. Nassor. Cronos: A separate compilation toolset for modular Esterel applications. In *Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, Toulouse, France, 1999.

[17] L. Hazard, J.-F. Susini, and F. Boussinot. The Junior reactive kernel. Research Report 3732, INRIA, July 1999.

[18] R. Lublinerman and S. Tripakis. Modularity vs. reusability: Code generation from synchronous block diagrams. In *Design, Automation and Test in Europe (DATE)*, 2008.

[19] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[20] K. Moessner, S. Hope, P. Cook, W. Tuttlebee, and R. Tafazolli. The RMA - a framework for reconfiguration of SDR equipement. *IEICE Trans. on Communications*, E85-B(12):2573–2580, December 2002.

[21] D. Potop-Butucaru and R. de Simone. Optimizations for faster execution of Esterel programs. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, Mont Saint-Michel, France, 2003.

[22] D. Potop-Butucaru, S. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.

[23] G. Rose. Authentication and security in mobile phones. In *Australian Unix Users Group conference, AUUG99*, Melbourne, Australia, September 1999.

[24] K. Schneider, J. Brandt, and E. Vecchié. Modular compilation of synchronous programs. In *IFIP Conference on Distributed and Parallel Embedded Systems (DIPES)*, Braga, Portugal, 2006.

[25] J.-P. Talpin and D. Nowak. A synchronous semantics of higher-order processes for modeling reconfigurable reactive systems. In *Proceedings of the 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 78–89, London, UK, 1998. Springer-Verlag.

[26] J. Zeng and S. Edwards. Separate compilation for synchronous modules. In *International Conference on Embedded Software and Systems (ICESS)*, Xian, China, 2005.